pSnort：基于多核处理器的并行入侵检测系统

贺鹏 姜海洋 谢高岗

摘 要： 网络入侵检测与防御系统在当前的 IP 网络安全领域中扮演着重要的角色，互联网流量的激增和单核处理器在数据包处理上存在的瓶颈，使得传统的运行于单核上的单线程网络入侵检测与防御系统已经远远不能满足网络发展的需求。为了解决这个问题，本文以主流单线程网络入侵检测与防御系统软件 Snort 为基础，设计了一个基于软件流水的并行入侵检测系统 pSnort，将传统的 Snort 划分为 2 个阶段，通过将其中最耗时的处理阶段并行化，以达到提升性能的目的。同时，通过程序设计，pSnort 避免了由于并行化而带来的严重的同步/互斥问题。经过试验，pSnort 在 Intel Quad-core Xeon 通用平台上可以获得超过 1Gbps 的包处理速度。相对于传统的 Snort，pSnort 最高能获得 147%的性能提升以及 2.5 倍加速比。

关键词： 多核 Snort 软流水 并行架构

1 引言

随着互联网的发展与广泛应用，缓冲区溢出攻击、拒绝服务攻击、后门漏洞攻击等各种网络攻击层出不穷。攻击者利用各种后门软件获取目标系统的管理员权限，向其他主机发动拒绝服务攻击。虽然主流的操作系统、应用软件定期发布各种安全补丁来修复软件中存在的漏洞，但其更新速度远远赶不上漏洞发现的速度，这使得网络入侵检测系统（Network Intrusion Detection System，NIDS）在网络安全领域显得越来越重要。

网络入侵检测系统是运行在网络边缘用于检测异常流量的网络安全设备。开源软件 Snort[1]作为主流网络入侵检测系统之一，运行于通用 CPU 平台上，使用特征选择的异常检测方式，通过定期更新异常特征数据库达到维护网络安全的目的。其他的网络入侵检测系统，例如 Bro 等基本上采用相类似的结构。当前，Snort 面临以下三方面的挑战：

- 在 Snort 系统中，异常特征数据库是以规则的形式存储的。这些规则刻画了异常流量的协议信息、端口信息和可疑内容等信息，数据包需要与规则集进行匹配。一般情况下，一种攻击对应着一条识别规则。随着网络中各种不同种类的攻击的增长，Snort 规则库中规则的数量也在不断增加。这使得 Snort 的检测计算负载加重。实际的测量结果表明，在单核平台上（2.0 GHz AMD Opteron 处理器），Snort 的吞吐量最高不到 500Mbps [4]；
- 攻击者利用目标操作系统 TCP/IP 协议栈的特点，将攻击代码以分片或者重叠的方式隐藏在不同包之间，以期望绕过网络入侵检测系统的检测。因此为了找出这些潜在的网络威胁，Snort 提供了 IP 分片重组、流重组等复杂功能还原网络中的数据流。这些功能所引起的计算与存储开销也是不小的；
- 芯片性能提高大致遵循摩尔定律，即每 18 个月性能提高一倍。而吉尔德定律认为，

在未来 25 年内，主干网的带宽将每 6 个月增加一倍。按照吉尔德定律的描述，带宽的增长指数超过摩尔定律所预测的 CPU 处理速度增长指数的 3 倍。因此，以传统 Snort 为代表的单线程网络入侵检测系统正面临着严重的性能难题。

多核处理器平台的出现为基于通用 CPU 平台实现的高速网络入侵检测系统带来了新的机遇。但传统网络入侵检测系统是针对单核平台开发的，Snort 以单线程模式运行，Bro 的数据分析部分也是单线程模式运行，它们不能充分利用多核处理器的特性提升性能。目前，多线程网络入侵检测系统正逐渐成为研究的热点。例如 Snort 开发小组正在开发新一代 Snort sp 3.0，以支持多线程数据包处理[5]；Bro 的主要开发人员也宣称多线程数据包处理将是他们下一步工作的重点[6]。然而网络数据包处理程序是一种有状态应用(Stateful Applications) [3]，其各模块间耦合比较紧密，数据依赖程度较高，并行化难度较大。以 Snort 为例，Snort 数据包处理虽然是逐包检测的，但 Snort 内部由于采用了 IP 分片重组和流重组机制，会联合处理属于同一个 IP 分片、同一个流的多个数据包，还原业务流以便进行深度数据包检测（DPI，Deep Packet Inspection）。在检测数据包时，Snort 必须依据业务流的上一个包所带来的状态改变，这使得基于包的并行策略非常难以应用于并行化 Snort。

针对这一现状，本文实验分析 Snort 各个检测过程的处理时间，在此基础上，提出了一种采用流并行机制与软件流水的并行入侵检测系统 pSnort。pSnort 着眼于并行化 Snort 处理流程中最耗时的部分以最大限度利用多核平台的计算资源，同时充分考虑到多线程可能带来的同步/互斥开销，从架构设计上将其予以消除。测试实验表明，pSnort 可以获得 1Gbps 的处理速度，相对于原始 Snort 可以获得最高 147%的性能提升，以及最高约 2.8 倍的加速比。

论文的其他章节安排如下：第二章主要介绍了并行网络入侵检测系统领域的一些相关工作，包括各种不同的并行执行方法介绍等；第三章根据 Snort 各个阶段的 CPU 处理时间，提出了并行架构的 pSnort 入侵检测系统；第四章描述了在 pSnort 具体实现的相关技术；第五章给出 pSnort 的实验测试结果和性能分析；第六章是结论与下一步工作的展望。

2 相关工作

传统网络入侵检测系统针对单核平台进行了充分的优化。然而近年来，由于 CPU 设计逻辑复杂，制造工艺与功耗受限[2]，性能不再能维持指数增长，业界开始转向多核平台以谋求性能提升。这导致了网络入侵检测系统并行化方法的研究热潮。各种针对当前多核平台的并行执行模型纷纷提出。在这些模型中，论文[3]总结了两种最基本的多线程执行模型：RTC(Run-To-Complete)和软件流水线(Software Pipeline, SPL)。

RTC 模型，顾名思义，即简单的复制多个执行单元，通过分配给不同的执行单元不同的任务，以达到性能提升的目的。每个线程都完成同样的功能：解码、分片重组、流查找和重组、规则匹配等步骤。每个核上绑定一个线程，这样核的数目增加就可以提升数据包处理的性能。

软件流水线模型将目标系统拆分为几个阶段，每个阶段作为一个单独的线程在一个核上完成。不同的线程之间通过队列或者缓冲池来传递消息。软件流水线模型借用了流水线的思想，提升目标系统的吞吐量。整体的处理流程在不同的核上分阶段完成，一个核只完成全部处理流程的一个部分，通过缓冲池将各个阶段的任务衔接起来。软件流水线模型的主要缺点在于增大了线程间通信的开销。在实际工程设计中，各个阶段的划分需要仔细的考量，以免

由多线程带来的同步/互斥开销给最终的性能提升带来严重影响。

哈格多任斯（B. Haagdorens）等人基于软流水的思想，提出了 5 种多线程 Snort 架构，并对这些架构的性能进行了评估。研究发现在双核并开启超线程的实验平台上（相当于四个逻辑核），5 种多线程架构中只有两种设计带来了 11%~16%的性能提升，其余 3 种设计反而带来了不同程度的性能下降[7]。该研究说明在采用软流水思想对 Snort 进行划分时，必须充分考虑到线程间同步的开销，否则这些开销所带来的性能损失往往会抵消由于多线程带来的性能收益。

英特尔在技术报告中也提出了利用多核技术提高 Snort 性能的方案[8]。在报告中，他们对完全并行和基于流分发的两种并行方案进行了吞吐量和二级缓存命中率的评估。他们的实验显示，在 Dual-Core Intel Xeon LV 处理器平台上，在数据包文件（trace）仅包含 175 个 TCP 连接的情况下，两者均可以达到 540~560Mbps 的吞吐量和 80%左右的缓存命中率，而在数据包文件包含 25000 个 TCP 连接时，基于流分发的并行方案可获得 188Mbps 的吞吐量和 70%的缓存命中率，完全并行方案只获得了 29Mbps 的吞吐量和 42%的缓存命中率。

为了实现更高速率(超过 10Gbps)的网络入侵检测系统，许多人提出基于硬件字符串匹配来加速深度数据包检测检测的方案。在这些方案中，TCAM[1]由于对最长前缀匹配（Longest Prefix Matching）的支持成为一种主流的硬件字符串匹配加速方案。余芳（音译，Yu Fang）等人将入侵检测系统字符串匹配划分为简单模式和复杂模式两大类，并将这两类模式都在 TCAM 上实现。仿真实验结果证明，该方案能提供 2Gbps 的处理能力[9]。成正石（音译，Jung-Sik Sung）等人改进了 TCAM 查找算法，他们的方案可以提供 10Gbps 的处理能力[10]。但是 TCAM 方案也有成本高、功耗高、存储效率不高等缺陷，而且也无法满足规则日益增长的要求[11]。

网络处理器（Network Processor）也是一种受到广泛关注的用于实现高性能网络入侵检测系统的平台。姜錫旼（音译， Seok-Min Kang）等人实现了一种基于网络处理器平台使用 TCAM 硬件技术的一个蠕虫病毒防御系统。该系统能够运行在 10Gbps 高速链路上[12]。但是使用网络处理器会使得编程难度增大，使用网络处理器的解决方案通常只适合靠近网络协议栈的底层的数据包处理，复杂的第四到第七层的业务处理往往会带来频繁的内存访问，此时使用网络处理器并不合适。

3 pSnort 架构

3.1 Snort 处理流程

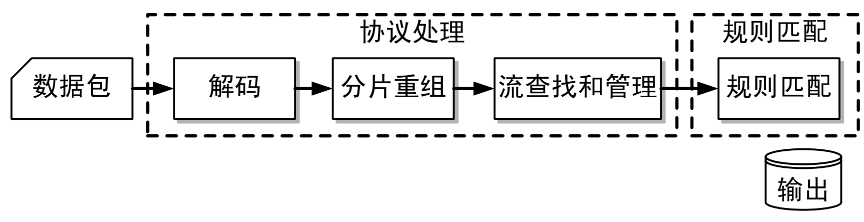


图1. Snort 处理流程

图 1 是一个典型的 Snort 数据包处理流程框图。其他的网络入侵检测系统也具有类似的

[1] Ternary Content Addressable Memory，三态内容可寻址存储器

流程。图中，Snort 数据包处理可以分为解码、预处理、规则分析、输出四个部分组成。其中预处理过程除包含分片重组、流记录管理、流重组等功能之外，还包含如端口扫描、HTTP2协议分析等功能。对于后端的规则检测模块来说，分片重组和流管理功能是必要的,其他功能则是可选的。在 Snort 代码内部，分片重组功能主要由 Frag3 模块实现，而流记录管理，包括流重组等功能主要由 Stream5 模块实现。整个预处理阶段，Frag3 和 Stream5 模块占用了绝大多数的处理时间。

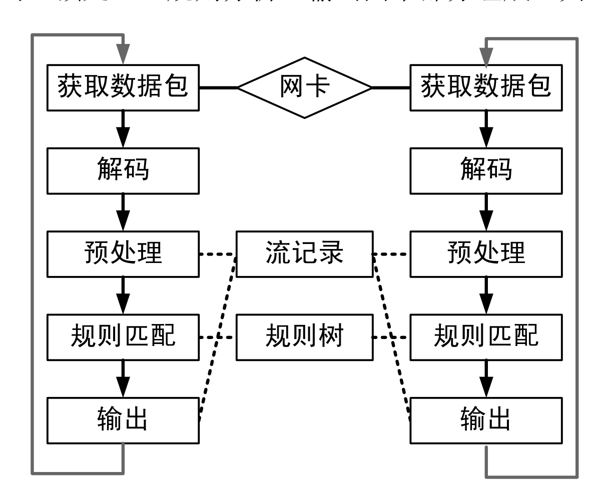


图2. RTC 并行化 Snort

采用 RTC 模型对 Snort 进行并行化设计时，需要尽量地降低由多线程所带来的同步/互斥开销。这些同步开销一般是由对共享数据的访问带来的。图 2 是 RTC 模型并行化 Snort 的一种架构设计。从图中可以看出，流记录的共享访问所带来的同步开销是很难避免的。当然，每个线程采用单独的流表的策略是可以避免这一开销的，但是这样一来，相关联的不同流可能被分配到不同的线程上处理，针对不同流进行的跨流检测便很难进行。

采用软件流水线模型对 Snort 进行并行化设计时，则需要避免将 Snort 各阶段划分得过细。由于当前通用 CPU 大多不支持硬件线程迁移和快速的核间通信，采用软件流水线模型的并行架构往往会带来很大的消息传递开销，从而影响最终性能。

3.2 pSnort 架构设计

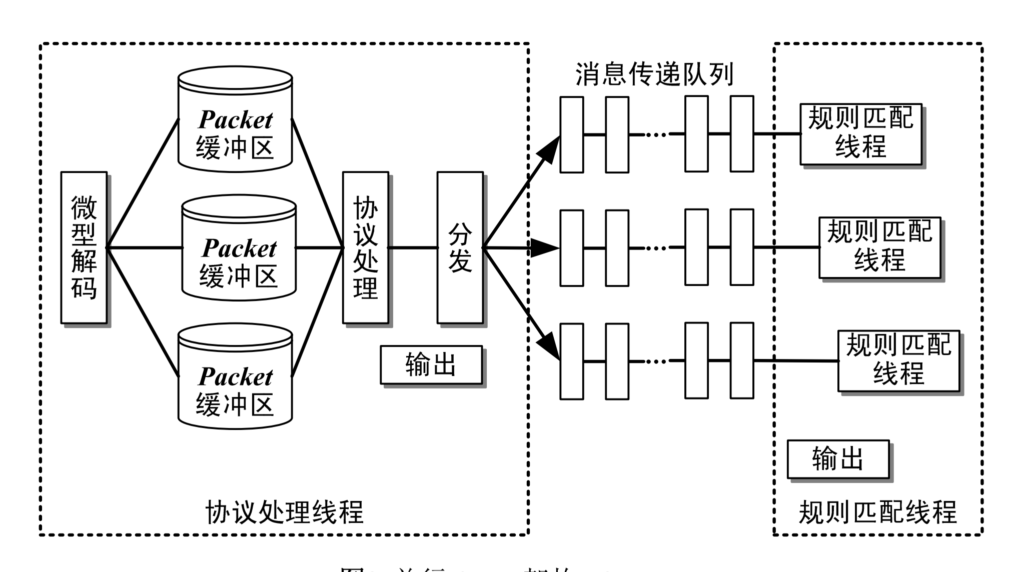


图3. 并行 Snort 架构 pSnort

根据图 1 所示，我们可以将 Snort 数据处理流程划分成两大阶段：（1）协议处理阶段；（2）规则匹配阶段。在协议处理阶段，Snort 关心的是数据包的协议头所携带的信息，如解

[2] HyperText Transfer Protocol，超文本传输协议

码阶段 Snort 完成的主要任务在于计算出数据包各层协议头在数据包中的位置，并保存下来为下阶段使用。在规则匹配阶段，Snort 关心的是数据包的负载。该阶段的主要任务在于依据规则数据库搜索数据包负载，判断该数据包是否可能携带各种攻击信息。

实验数据显示，在一台配有 Intel Xeon E5420 CPU、8G DDR2 667 内存的服务器上，协议处理阶段的处理时延为 3μs，规则匹配阶段的处理时延为 21μs[14]占处理时间的 87.5%，因为其中包含耗时的字符串匹配操作。很明显，规则匹配是整个 Snort 处理流程的瓶颈所在。因此，使用多个线程并行完成这一阶段任务，可以期望得到性能的提升。

根据上文的分析，我们提出了基于软件流水的并行 Snort 架构 pSnort，如图 3 所示。整个数据包处理由一个协议处理线程和多个规则匹配线程协作完成。由于协议处理线程只有一个，因此对于流记录的数据访问是不需要加锁的，这样就避免了多线程同时访问流记录所带来的互斥开销。而对于在规则匹配阶段需要多线程同时访问的规则数据库，因为所有的规则匹配操作对规则数据库只会进行读操作，所以规则匹配部分也是不需要加锁的。这样，我们就从架构设计上避免了多线程可能带来的严重同步/互斥开销。

尽管规则匹配是逐包进行的，但是本文提出的架构应用于 Snort 时仍不能使用基于包并行的策略。因为 Snort 规则描述语言中提供了 flowbit 关键字，该关键字必须在属于同一个流的包按序经过同一个规则匹配线程的情况下才能正常工作，否则可能有部分攻击会被漏掉。因此我们使用微型解码模块在协议处理模块之前，对数据包进行旨在获得四元组的少量解码工作，以保证属于相同的流的包会按序经过同一个规则匹配模块。

Snort 数据包处理代码使用了一个全局结构体 ***Packet***。每个数据包对应一个 ***Packet*** 结构体。由于 Snort 是逐包处理的，因此在整个处理流程中，只需要一个 ***Packet*** 实例即可。而在 pSnort 系统中，由于每个线程只完成全部处理流程的一部分，当线程处理完某一个数据包时，它会立刻去处理下一个数据包。这意味着在任意时刻，整个 pSnort 系统中会存在多个数据包被处理。因此需要一个 ***Packet*** 结构体的缓存池缓存多个 ***Packet*** 实例。

以下是 pSnort 从网络接口上获取一个数据包之后的处理流程：

1. 从缓冲池中获取一个 ***Packet*** 实例。进行协议处理。这阶段会给 ***Packet*** 结构体中各域赋予对应的地址值；
2. 通过分发模块将 ***Packet*** 实例放置到它所对应消息传递队列上；
3. 规则匹配线程则从各自对应的消息队列上取出 ***Packet*** 实例进行规则匹配。当匹配工作完成时，***Packet*** 实例会被放回到 ***Packet*** 缓存池中。

在协议处理阶段和规则匹配阶段都可能产生报警，因此，这些线程中都必须有一个输出模块。

3.3 数据结构设计

3.3.1 *Packet* 缓存池

Packet 缓存池在处理的两个阶段都需要使用到，因此对该数据结构需要使用锁保护。但是我们可以通过使用每个规则匹配线程对应一个 ***Packet*** 缓存池的方法缩小冲突域，从而降低锁争用出现的几率。由于每个规则匹配都有自己的 ***Packet*** 缓存池，属于同一个流的数据包又必须在同一个规则匹配线程上处理，在获取 ***Packet*** 实例前必须获得数据包的流信息。

如图 3 所示，在向 ***Packet*** 缓存池获取 ***Packet*** 实例之前，系统会对数据包进行微型解码，获取流信息，并根据流信息选择相应的 ***Packet*** 缓冲区。

Packet 缓冲池使用一个指针数组存储预先分配好的 ***Packet*** 实例，使用一个索引值完成 ***Packet*** 实例分配和放回的操作。其伪代码如图 4。

```
Process Get-Packet
     p=array_Packet[index];
    index++;
    return p;
Process Put-Packet(packet)
    index--;
    array_Packet[index]=packet;
```

图4. Packet 缓存伪代码

3.3.2 消息传递队列

每个规则匹配线程都包含一个消息传递队列，这样一来，协议处理线程和规则匹配线程之间构成了一个“单生产者-单消费者”关系。此时我们可以使用无锁的循环队列来实现消息传递队列。

由于消息队列是由无锁循环队列实现的，因此其队列长度需要预先设定。消息队列的合适长度取决于网络流量的突发情况，这是很难预先估计的。在本设计中，取值 1024 进行实验。当消息队列满，线程会不断重试，直至队列中有空闲的位置可以继续传递为止。

4 全局变量与数据依赖

并行化 Snort 需要解决两个问题：

– 全局变量私有化问题。即需要将某些全局变量变成线程私有的，以保证异常流量检测的正确进行；

– 数据依赖问题。

第一个问题使用 GCC[3]一个扩展属性__thread 可以很容易解决。

数据依赖问题是指由于消息传递队列，数据包在经过协议处理之后，规则匹配之前需要经历一段排队延迟。此时，规则匹配阶段所需要使用到的协议处理阶段的数据和信息可能是不一致的。例如，当一个 TCP 流中最后一个 FIN 包经过协议处理阶段后，该流所对应的流记录会被删除，但是此时该流可能会有一些包在消息传递队列中尚未被规则匹配阶段所处理。当规则匹配阶段处理这些包时，将无法找到这些包所对应的流记录，导致处理逻辑的混乱。

数据依赖问题可以划分为两个子问题：（1）当后端处理阶段需要使用前端处理阶段的信息时，该信息已经被更新；（2）当后端处理阶段需要使用前端处理阶段的信息时，该信息已经被删除。如果阶段划分可以做到完全的数据隔离，即不同的处理阶段需要的是不同的数据信息，则数据依赖问题即可避免。遗憾的是，Snort 并不符合这个条件。此外，在实际网络数据处理时，前后两个处理阶段处理速度不可能始终保持一致，因此数据依赖问题无法完全避免。

如果存在子问题（1），则协议处理阶段必须把每个包所对应的流状态信息记录下来，以

[3] GCC 是 GNU 的 C 和 C++ 编译器，是 Linux 中最重要的软件开发工具，GNU 是免费 Unix 风格编译器

备后端处理阶段查询。这将会消耗巨大的内存空间。如果只是存在子问题（2），则协议处理阶段只需要保存流记录直到可以确切删除的时候删除即可。幸运的是，Snort 处理本身不存在子问题（1）。因此我们采用了以下方法解决子问题（2）：

- 给每个流记录做引用计数，每当协议处理流程将数据包放入消息传递队列时，该数据包对应的流记录引用计数加一，每当规则匹配处理流程结束，该流记录引用计数减一。对引用计数的操作必须是原子的；
- 当协议处理流程需要删除流记录时，标记流记录的状态，同时将流记录从流表中取出，放入到一个删除链表中。该链表对于协议处理线程是不可见的，这样，该流记录不会影响到协议处理线程的正常处理，同时规则匹配线程也能够安全地访问到他所需要的流记录。当某一个流记录被标记为删除，同时它的引用计数为零，则可以安全地将该流记录从删除链表中删除。

5　实验验证和性能分析

5.1 实验平台

实验平台的硬件参数由表 1 给出。

表 1　实验环境的硬件参数

硬件名称	规格说明
CPU Intel Xeon E5420	2 路 4 核，共 8 核；每核一级缓存 32K: 16K(指令)+ 16K（数据）；四核两两共享二级缓存 6M
内存	DDR2 667 总共 8GB
网络接口	提供 DMA 功能的硬件加速卡

我们设计了一块硬件加速卡作为网络接口应用于 pSnort。该网卡使用直接内存映射，（Direct Memory Access）能够将数据包直接提供给程序。利用这块硬件加速卡，可以显著降低数据包捕获所带来的内存拷贝、CPU 中断以及用户/内核上下文切换所带来的开销，将 CPU 彻底从数据包捕获中解放出来，提高系统性能。该卡配有 4 个千兆端口，能同时处理最高 4Gbps 口，能同时处理最高 4Gbps 的流量，使用其中一个端口进行实验。测试数据包文件从某研究所出口网络 GE 链路捕获。表 2 显示了三个数据包文件的参数。

表 2 试验数据包文件的参数

数据包文件	平均包大小（字节）	大小（GB）
Trace1	662	1.4
Trace2	645	3.1
Trace 3	671	1.5

同时，通过 tcpreplay[13]在另一台服务器上发送数据包文件来模拟外部网络环境。实验中使用 tcpreplay 的-t 参数获取发送数据包的最高速率。在当前平台下，tcpreplay 能获得的最高发送速率为 918.18Mbps。

实验共选取了三个规则集，规则集 1 包含了 2217 条规则，规则集 2 包含了 2730 条规则，规则集 3 包含了 3258 条规则。

5.2 性能指标定义

实验中使用的性能指标为吞吐量和加速比。定义当有 $\boldsymbol{i}$ 个规则匹配线程开启时，$\boldsymbol{T_i}$ 为 pSnort 吞吐量（Mbps），$\boldsymbol{P_i}$ 为 pSnort 处理的包数目。$\boldsymbol{P_{total}}$ 表示数据包文件中数据包的个数，$\boldsymbol{S}$ 表示 tcpreplay 发送速率（Mbps）。吞吐量由以下公式得出：

$$T_i(\text{Mbps}) = \frac{P_i}{P_{total}} \times S(\text{Mpbs}) \tag{1}$$

定义 A_i 为 i 个规则匹配线程时的加速比，加速比由以下公式得出：

$$A_i = \frac{T_i(\text{Mpbs})}{T_1(\text{Mpbs})} \tag{2}$$

5.3 实验结果与分析

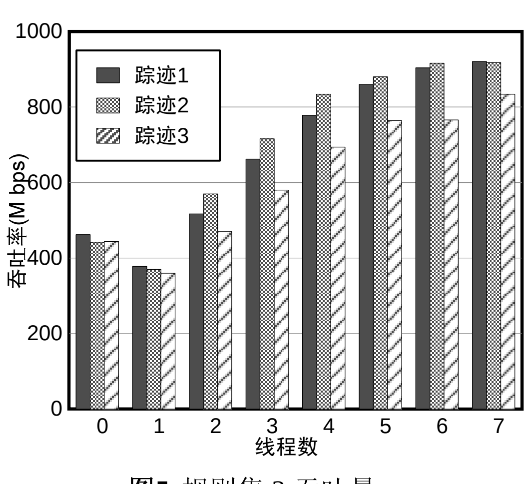


图5. 规则集 2 吞吐量

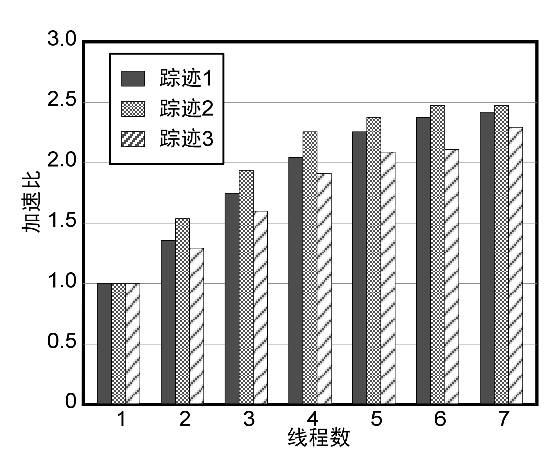


图6. 使用规则集 2 时的加速比

图 5 显示的是使用规则集 2 情况下，pSnort 吞吐量和线程数目之间的关系。线程数为 0 表示的是使用原始 Snort 所达到的吞吐量。从图中看出，当线程数为 1 时，由于 pSnort 与 Snort 相比，引入了锁开销，最终性能比原始的 Snort 要差。但是随着线程数目的增长，pSnort 的吞吐量也在增大。当线程数为 7 时，pSnort 可以完全处理 Trace1 和 Trace2 中的所有流量，达到当前平台下的最高吞吐率。

图 6 显示对应的加速比。

在三个不同数据包文件下，采用规则集 2 的 pSnort 相对于原始 Snort 分别获得 99%、107%、88%的性能提升。

在图中加速比没有线性增长，原因是我们的架构只并行了全部处理流程的一个部分。在各个线程的单包处理时延相等的情况下，加速比可以按照以下公式进行计算。

$$A_i = \frac{D_p + D_R}{D_p + \frac{D_R}{i}} \tag{3}$$

其中 D_p 表示协议处理阶段的单包处理时延， D_R 表示规则匹配阶段的单包处理时延。根据[14]，令 $D_p = 3$， $D_R = 21$， 可以得出理想加速比曲线。

由图可见，当 $i = 7$ 时，pSnort 的实际加速比与理想加速比相差 1.5 左右。这主要由以下三个方面的原因造成：

第一， 实际情况下流量不均衡。pSnort 使用的分流哈希算法不能保证分配到各个线程上的流量均衡。

第二， 实际上，线程数增多时，pSnort 前后两阶段的处理时延会慢慢增加，这进一步减缓了加速比。

第三， 由于匹配规则的特异性，少量的流会有大量的匹配规则对应（如 80 端口的 HTTP 流量）。即使各个线程处理的流量是均衡的，如果某个线程被分配到一个对应大量匹配规则的流，它的处理时延将增大，这导致各个线程的处理时延是不相等的。

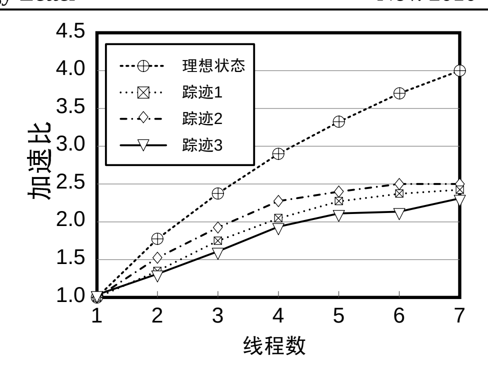


图7. 理想和实际加速比对比

根据公式（3），当 $D_p \ge D_R / t$ 时，协议处理阶段成为主要的性能瓶颈，此时，增大线程数不会带来显著的性能提升。令 $D_p = 3$， $D_R = 21$，那么当 $i \ge 7$ 时，协议处理阶段会称为性能瓶颈，增大线程数并不会带来显著的性能提升。尽管如此，我们可以通过改善均衡来提升 pSnort 的性能。

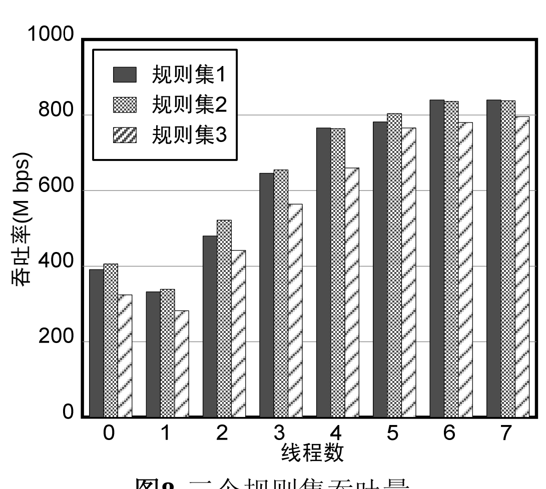


图8. 三个规则集吞吐量

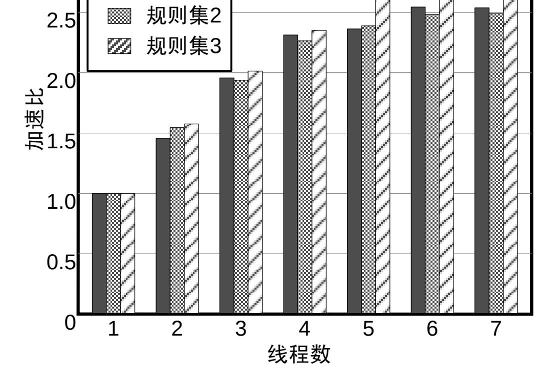


图9. 规则集加速比

图 8 显示不同规则集情况下，使用 Trace2 时，线程数和吞吐量之间的关系。线程数为 0 表示原始 Snort 获得的吞吐量。由于规则集 3 包含规则数比另两个规则集多，因此吞吐量有所下降。另外两个规则集均完全处理了 Trace2 中的所有数据包。

图 9 显示的是对应的加速比。可以看见由于规则数较多，匹配工作量加大，规则集 3 表现出更大的加速比。

使用 Trace 2，采用三个规则集的 pSnort 相对于原始 Snort 分别获得 115%、107%、147% 的性能提升。

6 结束语

为了充分利用多核平台所提供的计算能力，满足当前在高速链路下进行入侵检测的需求，本文提出了一种基于软件流水的 Snort 并行架构 pSnort。本架构充分考虑到由于多线程可能带来的同步/互斥问题，采用软件流水的方式，从架构上避免了流记录等共享数据的访问可能带来的锁争用开销，同时保留了对跨流检测的支持。实验证明，pSnort 相对于原始 Snort 最高可获得 147%的性能提升。由于网络处理程序大多具有相同或者相似的处理流程，我们所提出的架构是具有较高的通用性的。

下一步工作包括以下几个方面：

第一，本架构的实现只考虑了 Frag3 和 Stream5 两个预处理器，下一步考虑将其他预处理器在本架构下实现。

第二，优化 pSnort 的均衡算法，使 pSnort 获得更高的加速比。

第三，找出 pSnort 性能瓶颈，进行有针对性的优化。软件流水模型由于会传递数据而带来比较大的核间通信开销。当前的 CPU 架构特别是缓存系统普遍缺乏对核间通信的支持。我们期望通过仔细测量和分析 pSnort 性能参数，特别是缓存不命中的影响，来确定系统性能瓶颈，提出方法降低传递数据量来获得性能的提升。

第四，由于协议处理阶段大多是访存操作，计算量较小，对 CPU 主频要求并不高。我们可以设计出新的硬件加速卡，将协议处理阶段移植到硬件卡上自带的 CPU 中。这样能进一步减轻 CPU 负载，获得更高的性能。

参考文献：

[1] M. Roesch. Snort – Lightweight Intrusion Detection for Networks. In Proceedings of the 13th USENIX Conference on System Administration, pages 229–238, 1999.

[2] Gelsinger P. Microprocessors for the New Millennium: Challenges, Opportunities, and New Frontiers. Proceedings of the International Solid State Circuits Conference, 2001, 22-25.

[3] J Verdu, et al. MultiLayer Processing - An execution model for parallel stateful packet processing. ANCS'08, November 6-7, 2008, San Jose, CA, USA.

[4] Derek L. Schuff and Vijay S. Pai. Design Alternatives for a High-Performance Self-Securing Ethernet Network Interface. Parallel & Distributed Processing Symposium (IPDPS), 2007 IEEE International; Long Beach,CA .

[5] Snort Security Platform (SnortSP) 3.0 Beta http://www.snort.org/dl/snortsp, 2007

[6] V Paxson, R. Sommer. An Architecture for Exploiting Multi-Core Processors to Parallelize Network Instrusion Prevention. Proceedings of IEEE Sarnoff Symposium, 2007, 1-7.

[7] B Haagdorens, et al. Improving the performance of signature-based network intrusion detection sensors by multi-threading. Proceedings of the 5th International Workshop on Information Security Applications, 2004, 188-203.

[8] Intel. Supra-linear Packet Processing Performance with Intel® Multi-core Processors, 2006.

[9] Yu Fang, Randy H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In ICNP, pages 174–183, 2004.

[10] Jung-Sik Sung, eok Min Kang, Youngseok Lee, Taeck-Geun Kwon, and Bong-Tae Kim. A multi-gigabit rate deep packet inspection algorithm using tcam. In GLOCOM), pages 453–457, 2005.

[11] David E. Taylor. Survey and taxonomy of packet classification techniques. ACM Comput. Surv. 37(3):238–275, 2005.

[12] Seok-Min Kang et al. Design and Implementation of a Multi-gigabit Intrusion and Virus/Worm Detection System. ICC '06. IEEE International Conference on Communication. 2006.

[13] A.Turner. tcpreplay. http://tcpreplay.synfin.net/trac/

[14] Haiyang Jiang et. al. Performance Measurement and Analysis of Software Pipeline IDS Model on Multi-core Processor. Submitting to EuroNF Workshop on Traffic Management and Traffic Engineering for the Future Internet. 2009

作者简介：

贺　鹏： 中国科学院计算技术研究所，网络技术研究中心，博士在读生 hepeng@ict.ac.cn

姜海洋： 中国科学院计算技术研究所，网络技术研究中心，博士在读生

谢高岗： 中国科学院计算技术研究所，网络技术研究中心主任，研究员，博士生导师